

第8章 init 进程详解

在运行 Android 程序之后，首先会启动 init 进程，这个进程是 Linux 系统中用户空间的第一个进程，其进程号为 1。在本章的内容中，将详细讲解 Android 4.3 中 init 进程的运行过程，希望通过本章内容的学习，为读者步入本书后面高级知识的学习打下基础。

8.1 init 基础

我们知道，Android 本质上就是一个基于 Linux 内核的操作系统。与 Linux 的最大的区别是，Android 在应用层专门为移动设备添加了一些特有的支持。目前 Linux 有很多通信机制可以在用户空间和内核空间之间交互，例如设备驱动文件（位于“/dev”目录中）、内存文件（“/proc”、“/sys”目录等）。Android 在加载 Linux 基本内核后，就开始运行一个初始化进程 init。从 Android 加载 Linux 内核时设置了如下所示的参数。

```
Kernel command line: noinitrd root=/dev/nfs console=ttySAC0 init=/initnfsroot=192.168.1.103:/nfsbootip=192.168.1.20:192.168.1.103:192.168.1.1:255.255.255.0::eth0:on
```

在上述命令中，告诉 Linux 内核初始化完成后开始运行 init 进程，由于 init 进程就是放在系统根目录下面。而 init 进程的代码位于源码的目录“system/core/init”下面。在分析 init 的核心代码之前，还需要做如下所示的工作。

- 初始化属性。
- 处理配置文件的命令（主要是 init.rc 文件），包括处理各种 Action。
- 性能分析（使用 bootchart 工具）。
- 无限循环执行 command（启动其他的进程）。

init 程序并不是由一个源代码文件组成的，而是由一组源代码文件的目标文件链接而成的。这些文件位于如下所示的目录中：

```
/system/core/init
```

主要的 JNI 代码放在以下的路径中：

```
frameworks/base/core/jni/
```

另外，还涉及了其他目录中的以下文件：

```
\bionic\libc\bionic\libc_init_common.h  
\bionic\libc\bionic\libc_init_common.c  
\bionic\libc\bionic\libc_init_dynamic.c  
\bionic\libc\bionic\libc_init_static.c  
\system\core\libcutils\properties.c
```

在本章的内容中，将详细分析 init 进程的启动过程，了解 Android 系统是如何启动起来的。

8.2 分析入口函数

进程 init 入口函数是 main，具体实现文件的路径是：

system\core\init\init.c

函数 main 的实现非常复杂，从这个 main 函数可以看出，init 实际上就分为如下两个部分。

(1) 初始化。

初始化主要包括建立 “/dev” “/proc” 等目录，初始化属性，执行 init.rc 等初始化文件中的 action 等。

(2) 使用 for 循环、无限循环建立子进程。

这两项工作是 init 中的核心。在 Linux 系统中 init 是一切应用空间进程的父进程，因此平常在 Linux 终端执行命令，并建立进程，实际上都是在这个无限的 for 循环中完成的。也就是说，在 Linux 终端执行 ps-e 命令后，看到的所有除了 init 外的其他进程，都是由 init 负责创建的，而且 init 也会常驻内存。当然，如果 init 崩溃了，那么 Linux 系统就会基本上崩溃了。

函数 main 的具体实现代码如下所示：

```
int main(int argc, char **argv)
{
    int fd_count = 0;
    struct pollfd ufds[4];
    char *tmpdev;
    char* debuggable;
    char tmp[32];
    int property_set_fd_init = 0;
    int signal_fd_init = 0;
    int keychord_fd_init = 0;
    bool is_charger = false;

    if (!strcmp(basename(argv[0]), "ueventd"))
        return ueventd_main(argc, argv);

    if (!strcmp(basename(argv[0]), "watchdogd"))
        return watchdogd_main(argc, argv);

    /* clear the umask */
    umask(0);
    // 下面的代码开始建立各种用户空间的目录，如/dev、/proc、/sys 等
    mkdir("/dev", 0755);
    mkdir("/proc", 0755);
    mkdir("/sys", 0755);

    mount("tmpfs", "/dev", "tmpfs", MS_NOSUID, "mode=0755");
    mkdir("/dev/pts", 0755);
    mkdir("/dev/socket", 0755);
    mount("devpts", "/dev/pts", "devpts", 0, NULL);
    mount("proc", "/proc", "proc", 0, NULL);
    mount("sysfs", "/sys", "sysfs", 0, NULL);

    /* 检测/dev/.booting 文件是否可读写和创建*/
    close(open("/dev/.booting", O_WRONLY | O_CREAT, 0000));

    open_devnull_stdio();
    klog_init();
    // 初始化属性
    property_init();

    get_hardware_name(hardware, &revision);
    // 处理内核命令行
    process_kernel_cmdline();
    ...
}
```

```

is_charger = !strcmp(bootmode, "charger");

INFO("property init\n");
if (!is_charger)
    property_load_boot_defaults();

INFO("reading config file\n");
// 分析/init.rc 文件的内容
init_parse_config_file("/init.rc");
... ... // 执行初始化文件中的动作
action_for_each_trigger("init", action_add_queue_tail);
// 在 charger 模式下略过mount 文件系统的工作
if (!is_charger) {
    action_for_each_trigger("early-fs", action_add_queue_tail);
    action_for_each_trigger("fs", action_add_queue_tail);
    action_for_each_trigger("post-fs", action_add_queue_tail);
    action_for_each_trigger("post-fs-data", action_add_queue_tail);
}

queue_builtin_action(property_service_init_action, "property_service_init");
queue_builtin_action(signal_init_action, "signal_init");
queue_builtin_action(check_startup_action, "check_startup");

if (is_charger) {
    action_for_each_trigger("charger", action_add_queue_tail);
} else {
    action_for_each_trigger("early-boot", action_add_queue_tail);
    action_for_each_trigger("boot", action_add_queue_tail);
}

/* run all property triggers based on current state of the properties */
queue_builtin_action(queue_property_triggers_action, "queue_property_triggers");

#endif BOOTCHART
queue_builtin_action(bootchart_init_action, "bootchart_init");
#endif
// 进入无限循环, 建立 init 的子进程 ( init 是所有进程的父进程 )
for(;;) {
    int nr, i, timeout = -1;
    // 执行命令 (子进程对应的命令)
    execute_one_command();
    restart_processes();

    if (!property_set_fd_init && get_property_set_fd() > 0) {
        ufds[fd_count].fd = get_property_set_fd();
        ufds[fd_count].events = POLLIN;
        ufds[fd_count].revents = 0;
        fd_count++;
        property_set_fd_init = 1;
    }
    if (!signal_fd_init && get_signal_fd() > 0) {
        ufds[fd_count].fd = get_signal_fd();
        ufds[fd_count].events = POLLIN;
        ufds[fd_count].revents = 0;
        fd_count++;
        signal_fd_init = 1;
    }
    if (!keychord_fd_init && get_keychord_fd() > 0) {
        ufds[fd_count].fd = get_keychord_fd();
        ufds[fd_count].events = POLLIN;
        ufds[fd_count].revents = 0;
        fd_count++;
        keychord_fd_init = 1;
    }

    if (process_needs_restart) {
        timeout = (process_needs_restart - gettimeofday()) * 1000;
    }
}

```

```

        if (timeout < 0)
            timeout = 0;
    }

    if (!action_queue_empty() || cur_action)
        timeout = 0;
// bootchart 是一个性能统计工具，用于搜集硬件和系统的信息，并将其写入磁盘，以便其
// 他程序使用
#endif BOOTCHART
    if (bootchart_count > 0) {
        if (timeout < 0 || timeout > BOOTCHART_POLLING_MS)
            timeout = BOOTCHART_POLLING_MS;
        if (bootchart_step() < 0 || --bootchart_count == 0) {
            bootchart_finish();
            bootchart_count = 0;
        }
    }
#endif
// 等待下一个命令的提交
nr = poll(ufds, fd_count, timeout);
if (nr <= 0)
    continue;

for (i = 0; i < fd_count; i++) {
    if (ufds[i].revents == POLLIN) {
        if (ufds[i].fd == get_property_set_fd())
            handle_property_set_fd();
        else if (ufds[i].fd == get_keychord_fd())
            handle_keychord();
        else if (ufds[i].fd == get_signal_fd())
            handle_signal();
    }
}
}

return 0;
}

```

8.3 配置文件详解

在 init 进程中，配置文件是指文件 init.rc，其路径是：

\system\core\rootdir\init.rc

在本节的内容中，将详细剖析讲解配置文件 init.rc 的基本知识。

8.3.1 init.rc 简介

文件 init.rc 是一个可配置的初始化文件，在里面定制了厂商可以配置的额外的初始化配置信息，具体格式为：

init.%PRODUCT%.rc

文件 init.rc 是在文件 “/system/core/init/init.c” 中读取的，它基于“行”，包含一些用空格隔开的关键字（它属于特殊字符）。如果在关键字中含有空格，则使用 “/” 表示转义，使用 “” 防止关键字被断开，如果 “/” 在末尾则表示换行，以 “#” 开头的表示注释。

文件 init.rc 包含 4 种状态类别，分别是 Actions、Commands、Services 和 Options，当声明一个 Service 或者 Action 的时候，它将隐式声明一个 section，它之后跟随的 Command 或者 Option 都将属于这个 section。另外，Action 和 Service 不能重名，否则忽略为 error。

(1) Actions。

Actions 就是在某种条件下触发一系列的命令，通常有一个 trigger，形式如下所示：

```
on <trigger>
  <command>
  <command>
```

(2) Service。

Service 的结构如下所示：

```
service <name> <pathname> [ <argument> ]*
  <option>
  <option>
```

(3) Option。

Option 是 Service 的修饰词，主要包括如下所示的选项。

- critical：表示如果服务在 4 分钟内存在多于 4 次，则系统重启到 recovery mode。
- disabled：表示服务不会自动启动，需要手动调用名字启动。
- setEnv <name> <value>：设置启动环境变量。
- socket <name> <type> <permission> [<user> [<group>]]：开启一个 unix 域的 socket，名字为 /dev/socket/<name>，<type>只能是 dgram 或者 stream，<user>和<group>默认为 0。
- user <username>：表示将用户切换为<username>，用户名已经定义好了，只能是 system/root。
- group <groupname>：表示将组切换为<groupname>。
- oneshot：表示这个 Service 只启动一次。
- class <name>：指定一个要启动的类，这个类中如果有多个 service，将会被同时启动。默认的 class 将会是“default”。
- onrestart：在重启时执行一条命令。

(4) trigger。

主要包括如下所示的选项。

- boot：当 /init.conf 加载完毕时。
- <name>=<value>：当<name>被设置为<value>时。
- device-added-<path>：设备<path>被添加时。
- device-removed-<path>：设备<path>被移除时。
- service-exited-<name>：服务<name>退出时。

(5) 主要包含的操作命令及具体说明。

- exec <path> [<argument>]*：执行一个<path>指定的程序。
- export <name> <value>：设置一个全局变量。
- ifup <interface>：使网络接口<interface>连接。
- import <filename>：引入其他的配置文件。
- hostname <name>：设置主机名。
- chdir <directory>：切换工作目录。
- chmod <octal-mode> <path>：设置访问权限。
- chown <owner> <group> <path>：设置用户和组。
- chroot <directory>：设置根目录
- class_start <serviceclass>：启动类中的 service。
- class_stop <serviceclass>：停止类中的 service。
- domainname <name>：设置域名。
- insmod <path>：安装模块。
- mkdir <path> [mode] [owner] [group]：创建一个目录，并可以指定权限，用户和组。

- mount <type> <device> <dir> [<mountoption>]*: 加载指定设备到目录下。
- <mountoption>: 包括"ro", "rw", "remount", "noatime"。
- setprop <name> <value>: 设置系统属性。
- setrlimit <resource> <cur> <max>: 设置资源访问权限。
- start <service>: 开启服务。
- stop <service>: 停止服务。
- symlink <target> <path>: 创建一个动态链接。
- sysclktz <mins_west_of_gmt>: 设置系统时钟。
- trigger <event>: 触发事件。
- write <path> <string> [<string>]*: 向<path>路径的文件写入多个<string>。

读者可以进入到 Android 的 shell，会看到根目录有一个 init.rc 文件。启动 Android 后，会将文件 init.rc 装载到内存。而修改文件 init.rc 的内容实际上只是修改内存中的 init.rc 文件的内容。一旦重启 Android，文件 init.rc 的内容又会恢复到最初的装载。想彻底修改文件 init.rc 内容，唯一方法是修改 Android 的 ROM 中的内核镜像（boot.img）。

8.3.2 分析 init.rc 的过程

文件 init.rc 是一个配置文件，内部有许多的语言规则，所有语言会在函数 init_parse_config_file 中进行解析。前面的主函数 main 读取完配置文件 init.rc 后，会调用函数 parse_config 进行解析。整个实现流程如下所示：

init_parse_config_file->read_file->parse_config

(1) 函数 parse_config 和 init_parse_config_file 在如下文件中实现：

\system\core\init\init_parser.c

函数 parse_config 和 init_parse_config_file 的具体实现代码如下所示：

```
static void parse_config(const char *fn, char *s)//s为init.rc中字符串的内容
{
    struct parse_state state;
    char *args[INIT_PARSER_MAXARGS];
    int nargs;

    nargs = 0;
    state.filename = fn;
    state.line = 1;
    state.ptr = s;
    state.nexttoken = 0;
    state.parse_line = parse_line_no_op;
    for (;;) {
        switch (next_token(&state)) {
        case T_EOF: //文件的结尾
            state.parse_line(&state, 0, 0);
            return;
        case T_NEWLINE://新的一行
            if (nargs) {
                int kw = lookup_keyword(args[0]); //读取 init.rc 返回关键字例如 service,返回 K_service
                if (kw_is(kw, SECTION)) { //查看关键字是否为 SECTION,只有 service 和 on 满足
                    state.parse_line(&state, 0, 0);
                    parse_new_section(&state, kw, nargs, args);
                } else {
                    state.parse_line(&state, nargs, args); //on 和 service 两个段下面的内容
                }
                nargs = 0;
            }
            break;
        case T_TEXT://文本内容
```

```

        if (nargs < INIT_PARSER_MAXARGS) {
            args[nargs++] = state.text;
        }
        break;
    }
}

int init_parse_config_file(const char *fn)
{
    char *data;
    data = read_file(fn, 0);
    if (!data) return -1;

    parse_config(fn, data);
    DUMP();
    return 0;
}

```

通过上述代码可以看出，在 for 的无限循环中对文件 init.rc 的内容进行了解析，以一行一行的形式进行了读取。

(2) 每读取完一行内容换行时到下一行时，使用函数 lookup_keyword 分析已经读取的一行的第一个参数。函数 lookup_keyword 的具体实现代码如下所示：

```

int lookup_keyword(const char *s)
{
    switch (*s++) {
    case 'c':
        if (!strcmp(s, "copy")) return K_copy;
        if (!strcmp(s, "apability")) return K_capability;
        if (!strcmp(s, "chdir")) return K_chdir;
        if (!strcmp(s, "hroot")) return K_chroot;
        if (!strcmp(s, "lass")) return K_class;
        if (!strcmp(s, "lass_start")) return K_class_start;
        if (!strcmp(s, "lass_stop")) return K_class_stop;
        if (!strcmp(s, "lass_reset")) return K_class_reset;
        if (!strcmp(s, "onsole")) return K_console;
        if (!strcmp(s, "hown")) return K_chown;
        if (!strcmp(s, "hmod")) return K_chmod;
        if (!strcmp(s, "ritical")) return K_critical;
        break;
    case 'd':
        if (!strcmp(s, "isabled")) return K_disabled;
        if (!strcmp(s, "omainname")) return K_domainname;
        break;
    case 'e':
        if (!strcmp(s, "xec")) return K_exec;
        if (!strcmp(s, "xport")) return K_export;
        break;
    case 'g':
        if (!strcmp(s, "roup")) return K_group;
        break;
    case 'h':
        if (!strcmp(s, "ostname")) return K_hostname;
        break;
    case 'i':
        if (!strcmp(s, "oprio")) return K_ioprio;
        if (!strcmp(s, "fup")) return K_ifup;
        if (!strcmp(s, "nsmod")) return K_insmod;
        if (!strcmp(s, "mport")) return K_import;
        break;
    case 'k':
        if (!strcmp(s, "eycodes")) return K_keycodes;
        break;
    case 'l':
        if (!strcmp(s, "oglevel")) return K_loglevel;
        if (!strcmp(s, "oad_persist_props")) return K_load_persist_props;
        break;
    }
}

```

```

    case 'm':
        if (!strcmp(s, "kdir")) return K_mkdir;
        if (!strcmp(s, "ount_all")) return K_mount_all;
        if (!strcmp(s, "ount")) return K_mount;
        break;
    case 'o':
        if (!strcmp(s, "n")) return K_on;
        if (!strcmp(s, "neshot")) return K_oneshot;
        if (!strcmp(s, "nrestart")) return K_onrestart;
        break;
    case 'r':
        if (!strcmp(s, "estart")) return K_restart;
        if (!strcmp(s, "estorecon")) return K_restorecon;
        if (!strcmp(s, "mdir")) return K_rmdir;
        if (!strcmp(s, "m")) return K_rm;
        break;
    case 's':
        if (!strcmp(s, "eclabel")) return K_seclabel;
        if (!strcmp(s, "ervice")) return K_service;
        if (!strcmp(s, "etcon")) return K_setcon;
        if (!strcmp(s, "etenforce")) return K_setenforce;
        if (!strcmp(s, "etenv")) return K_setenv;
        if (!strcmp(s, "etkey")) return K_setkey;
        if (!strcmp(s, "etprop")) return K_setprop;
        if (!strcmp(s, "etrlimit")) return K_setrlimit;
        if (!strcmp(s, "etsebool")) return K_setsebool;
        if (!strcmp(s, "ocket")) return K_socket;
        if (!strcmp(s, "tart")) return K_start;
        if (!strcmp(s, "top")) return K_stop;
        if (!strcmp(s, "ymlink")) return K_symlink;
        if (!strcmp(s, "ysclktz")) return K_sysclktz;
        break;
    case 't':
        if (!strcmp(s, "rigger")) return K_trigger;
        break;
    case 'u':
        if (!strcmp(s, "ser")) return K_user;
        break;
    case 'w':
        if (!strcmp(s, "rite")) return K_write;
        if (!strcmp(s, "ait")) return K_wait;
        break;
    }
    return K_UNKNOWN;
}

```

由此可见，函数 `lookup_keyword` 主要对每一行的第一个字符做 `case` 判断，然后在 `if` 语句中调用 `strcmp` 命令，这些命令都是按照文件 `init.rc` 的格式要求进行的。比如常用的 `service` 和 `on` 等经过 `lookup_keyword` 后返回 `K_servcie` 和 `K_on`。随后使用 `kw_is (kw, SECTION)` 判断返回的 `kw` 是否属于 `SECTION` 类型。在文件 `init.rc` 中，只有 `service` 和 `on` 满足该类型。

(3) 定义关键字。

在文件 `keywords.h` 中定义了 `init` 使用的关键字，在此文件中定义了如 `do_class_start`、`do_class_stop` 之类的函数，并且还定义了枚举。文件 `keywords.h` 的路径如下所示：

```
\system\core\init\
```

文件 `keywords.h` 的具体实现代码如下所示：

```

#ifndef KEYWORD
int do_chroot(int nargs, char **args);
int do_chdir(int nargs, char **args);
int do_class_start(int nargs, char **args);
int do_class_stop(int nargs, char **args);
int do_class_reset(int nargs, char **args);
.....
#define __MAKE_KEYWORD_ENUM__

```

```

#define KEYWORD(symbol, flags, nargs, func) K_##symbol,
enum {
    K_UNKNOWN,
#endif
    KEYWORD(capability, OPTION, 0, 0)
    KEYWORD(chdir,      COMMAND, 1, do_chdir)
    KEYWORD(chroot,     COMMAND, 1, do_chroot)
    KEYWORD(class,      OPTION, 0, 0)
    KEYWORD(class_start, COMMAND, 1, do_class_start)
    KEYWORD(class_stop,  COMMAND, 1, do_class_stop)
    KEYWORD(class_reset, COMMAND, 1, do_class_reset)
    KEYWORD(console,    OPTION, 0, 0)
    KEYWORD(critical,   OPTION, 0, 0)
    KEYWORD(disabled,   OPTION, 0, 0)
    KEYWORD(domainname, COMMAND, 1, do_domainname)
    KEYWORD(exec,       COMMAND, 1, do_exec)
    KEYWORD(export,     COMMAND, 2, do_export)

.....
    KEYWORD(load_persist_props,   COMMAND, 0, do_load_persist_props)
    KEYWORD(ioprio,        OPTION, 0, 0)
#endif _MAKE_KEYWORD_ENUM_
    KEYWORD_COUNT,
};

#undef _MAKE_KEYWORD_ENUM_
#undef KEYWORD
#endif

```

文件 keywords.h 在文件 init_parser.c 中被用到了两次，具体引用代码如下所示：

```

#define SECTION 0x01
#define COMMAND 0x02
#define OPTION 0x04

#include "keywords.h"

#define KEYWORD(symbol, flags, nargs, func) \
[ K_##symbol ] = { #symbol, func, nargs + 1, flags, },

struct {
    const char *name;
    int (*func)(int nargs, char **args);
    unsigned char nargs;
    unsigned char flags;
} keyword_info[KEYWORD_COUNT] = {
    [ K_UNKNOWN ] = { "unknown", 0, 0, 0 },
#include "keywords.h"
};

#undef KEYWORD

#define kw_is(kw, type) (keyword_info[kw].flags & (type))
#define kw_name(kw) (keyword_info[kw].name)
#define kw_func(kw) (keyword_info[kw].func)
#define kw_nargs(kw) (keyword_info[kw].nargs)

```

8.4

解析 service

由前面的函数 lookup_keyword 可知，在调用过程中会对 on 和 service 所在的段进行解析，这里首先分析 service，在分析时以文件 init.rc 中的 service zygote 为例。

8.4.1 Zygote 对应的 service action

在文件 init.rc 中，zygote 对应的 service action 的代码如下所示：

```

service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
    class main
    socket zygote stream 660 root system

```

```

onrestart write /sys/android_power/request_state wake
onrestart write /sys/power/state on
onrestart restart media
onrestart restart netd

```

解析 action 的入口函数是 parse_new_section，在此函数中再分别对 service 或者 on 关键字开头的内容进行解析。函数 parse_new_section 的具体实现代码如下所示：

```

void parse_new_section(struct parse_state *state, int kw,
                      int nargs, char **args)
{
    printf("[ %s %s ]\n", args[0],
           nargs > 1 ? args[1] : "");
    switch(kw) {
    case K_service:
        state->context = parse_service(state, nargs, args);
        if (state->context) {
            state->parse_line = parse_line_service;
            return;
        }
        break;
    case K_on:
        state->context = parse_action(state, nargs, args);
        if (state->context) {
            state->parse_line = parse_line_action;
            return;
        }
        break;
    case K_import:
        parse_import(state, nargs, args);
        break;
    }
    state->parse_line = parse_line_no_op;
}

```

8.4.2 init 组织 service

在 init 进程中，使用了一个名为 service 的结构体保存和 service action 有关的信息。此结构体是在如下文件中中定义的：

\system\core\init\init.h

结构体 service 的具体实现代码如下所示：

```

struct service {
    /* list of all services */
    struct listnode slist;

    const char *name;
    const char *classname;

    unsigned flags;
    pid_t pid;
    time_t time_started; /* time of last start */
    time_t time_crashed; /* first crash within inspection window */
    int nr_crashed; /* number of times crashed within window */

    uid_t uid;
    gid_t gid;
    gid_t supp_gids[NR_SVC_SUPP_GIDS];
    size_t nr_supp_gids;

#ifdef HAVE_SELINUX
    char *seclabel;
#endif

    struct socketinfo *sockets;
    struct svcenvinfo *envvars;
}

```

```

struct action onrestart; /* Actions to execute on restart. */

/* keycodes for triggering this service via /dev/keychord */
int *keycodes;
int nkeycodes;
int keychord_id;

int ioprio_class;
int ioprio_pri;

int nargs;
/* "MUST BE AT THE END OF THE STRUCT" */
char *args[1];
}; /* ^-----'args' MUST be at the end of this struct! */

```

另外，在文件 init.h 中还定义了结构体 action，具体实现代码如下所示：

```

struct action {
    /* node in list of all actions */
    struct listnode alist;
    /* node in the queue of pending actions */
    struct listnode qlist;
    /* node in list of actions for a trigger */
    struct listnode tlist;

    unsigned hash;
    const char *name;

    struct listnode commands;
    struct command *current;
};

```

这样便通过上述两个结构体对 service 进行了组织。

8.4.3 函数 parse_service 和 parse_line_service

在解析 Service 会用到两个函数，分别是 parse_service 和 parse_line_service。

(1) 当解析文件 init.rc 中的 service zygote 时会执行函数 parse_service，此函数的功能是构建 service 的骨架，对 service 关键字开头的内容进行解析。函数 parse_service 的具体实现代码如下所示：

```

static void *parse_service(struct parse_state *state, int nargs, char **args)
{
    struct service *svc;
    if (nargs < 3) {
        parse_error(state, "services must have a name and a program\n");
        return 0;
    }
    if (!valid_name(args[1])) {
        parse_error(state, "invalid service name '%s'\n", args[1]);
        return 0;
    }

    svc = service_find_by_name(args[1]); //查找服务是否存在
    if (svc) {
        parse_error(state, "ignored duplicate definition of service '%s'\n", args[1]);
        return 0;
    }

    nargs -= 2;
    svc = calloc(1, sizeof(*svc) + sizeof(char*) * nargs);
    if (!svc) {
        parse_error(state, "out of memory\n");
        return 0;
    }
    svc->name = args[1];           //sevice 的名字
    svc->classname = "default"; //svc 的类名默认是 default
    memcpy(svc->args, args + 2, sizeof(char*) * nargs); //首个参数放的是可执行文件
}

```

```

    svc->args[nargs] = 0;
    svc->nargs = nargs;//参数个数
    svc->onrestart.name = "onrestart";
    list_init(&svc->onrestart.commands);
    list_add_tail(&service_list, &svc->slist);
    return svc;
}

```

在上述代码中，args[1]就是 zygote，系统会先查找是否存在该服务，然后构建一个 service svc 并进行相关的填充，包括服务名、服务所属的类别名字和已经服务启动带入的参数个数（要减去 service 和服务名 zygote），最后将这个 svc 加入到 service_list 全局链表中。

(2) 函数 parse_line_service 的功能是根据配置文件的内容填充 service 结构体，并解析 Service 中剩余行中的 Option，比如 class、socket、onrestart 等。函数 parse_line_service 的具体实现代码如下所示：

```

static void parse_line_service(struct parse_state *state, int nargs, char **args)
{
    struct service *svc = state->context;
    struct command *cmd;
    int i, kw, kw_nargs;

    if (nargs == 0) {
        return;
    }

    svc->ioprio_class = IoSchedClass_NONE;

    kw = lookup_keyword(args[0]);
    switch (kw) {
    case K_capability:
        break;
    case K_class:
        if (nargs != 2) {
            parse_error(state, "class option requires a classname\n");
        } else {
            svc->classname = args[1];
        }
        break;
    case K_console:
        svc->flags |= SVC_CONSOLE;
        break;
    case K_disabled:
        svc->flags |= SVC_DISABLED;
        svc->flags |= SVC_RC_DISABLED;
        break;
    case K_ioprio:
        if (nargs != 3) {
            parse_error(state, "ioprio optin usage: ioprio <rt|be|idle> <ioprio 0-7>\n");
        } else {
            svc->ioprio_pri = strtoul(args[2], 0, 8);

            if (svc->ioprio_pri < 0 || svc->ioprio_pri > 7) {
                parse_error(state, "priority value must be range 0 - 7\n");
                break;
            }

            if (!strcmp(args[1], "rt")) {
                svc->ioprio_class = IoSchedClass_RT;
            } else if (!strcmp(args[1], "be")) {
                svc->ioprio_class = IoSchedClass_BE;
            } else if (!strcmp(args[1], "idle")) {
                svc->ioprio_class = IoSchedClass_IDLE;
            } else {
                parse_error(state, "ioprio option usage: ioprio <rt|be|idle> <0-7>\n");
            }
        }
    }
}

```

```

        break;
case K_group:
    if (nargs < 2) {
        parse_error(state, "group option requires a group id\n");
    } else if (nargs > NR_SVC_SUPP_GIDS + 2) {
        parse_error(state, "group option accepts at most %d supp. groups\n",
                    NR_SVC_SUPP_GIDS);
    } else {
        int n;
        svc->gid = decode_uid(args[1]);
        for (n = 2; n < nargs; n++) {
            svc->supp_gids[n-2] = decode_uid(args[n]);
        }
        svc->nr_supp_gids = n - 2;
    }
    break;
case K_keycodes:
    if (nargs < 2) {
        parse_error(state, "keycodes option requires atleast one keycode\n");
    } else {
        svc->keycodes = malloc((nargs - 1) * sizeof(svc->keycodes[0]));
        if (!svc->keycodes) {
            parse_error(state, "could not allocate keycodes\n");
        } else {
            svc->nkeycodes = nargs - 1;
            for (i = 1; i < nargs; i++) {
                svc->keycodes[i - 1] = atoi(args[i]);
            }
        }
    }
    break;
case K_oneshot:
    svc->flags |= SVC_ONESHOT;
    break;
case K_onrestart:
    nargs--;
    args++;
    kw = lookup_keyword(args[0]);
    if (!kw_is(kw, COMMAND)) {
        parse_error(state, "invalid command '%s'\n", args[0]);
        break;
    }
    kw_nargs = kw_nargs(kw);
    if (nargs < kw_nargs) {
        parse_error(state, "%s requires %d %s\n", args[0], kw_nargs - 1,
                    kw_nargs > 2 ? "arguments" : "argument");
        break;
    }

    cmd = malloc(sizeof(*cmd) + sizeof(char*) * nargs);
    cmd->func = kw_func(kw);
    cmd->nargs = nargs;
    memcpy(cmd->args, args, sizeof(char*) * nargs);
    list_add_tail(&svc->onrestart.commands, &cmd->clist);
    break;
case K_critical:
    svc->flags |= SVC_CRITICAL;
    break;
case K_setenv: { /* name value */
    struct svccenvinfo *ei;
    if (nargs < 2) {
        parse_error(state, "setenv option requires name and value arguments\n");
        break;
    }
    ei = calloc(1, sizeof(*ei));
    if (!ei) {
        parse_error(state, "out of memory\n");
        break;
    }
}

```

```

ei->name = args[1];
ei->value = args[2];
ei->next = svc->envvars;
svc->envvars = ei;
break;
}
case K_socket: /* name type perm [ uid gid ] */
    struct socketinfo *si;
    if (nargs < 4) {
        parse_error(state, "socket option requires name, type, perm arguments\n");
        break;
    }
    if (strcmp(args[2],"dgram") && strcmp(args[2],"stream")
        && strcmp(args[2],"seqpacket")) {
        parse_error(state, "socket type must be 'dgram', 'stream' or 'seqpacket'\n");
        break;
    }
    si = calloc(1, sizeof(*si));
    if (!si) {
        parse_error(state, "out of memory\n");
        break;
    }
    si->name = args[1];
    si->type = args[2];
    si->perm = strtoul(args[3], 0, 8);
    if (nargs > 4)
        si->uid = decode_uid(args[4]);
    if (nargs > 5)
        si->gid = decode_uid(args[5]);
    si->next = svc->sockets;
    svc->sockets = si;
    break;
}
case K_user:
    if (nargs != 2) {
        parse_error(state, "user option requires a user id\n");
    } else {
        svc->uid = decode_uid(args[1]);
    }
    break;
case K_seclabel:
#endif HAVE_SELINUX
    if (nargs != 2) {
        parse_error(state, "seclabel option requires a label string\n");
    } else {
        svc->seclabel = args[1];
    }
#endif
    break;

default:
    parse_error(state, "invalid option '%s'\n", args[0]);
}
}

```

8.5

字段 on

在本节的内容中，将详细剖析 on 字段的内容，并以 on boot 这个 section 作为例子进行分析。希望读者认真学习，为步入本书后面知识的学习打下基础。

8.5.1 Zygote 对应的 on action

字段 on 的内容比较复杂，此处将以 on boot 这个 section 作为例子进行分析。在文件 init.rc 中，zygote 对应的 on boot action 的代码如下所示：

```

on boot
# basic network init
ifup lo
hostname localhost
domainname localdomain

# set RLIMIT_NICE to allow priorities from 19 to -20
setrlimit 13 40 40

# Memory management. Basic kernel parameters, and allow the high
# level system server to be able to adjust the kernel OOM driver
# parameters to match how it is managing things.
write /proc/sys/vm/overcommit_memory 1
write /proc/sys/vm/min_free_order_shift 4
chown root system /sys/module/lowmemorykiller/parameters/adj
chmod 0664 /sys/module/lowmemorykiller/parameters/adj
chown root system /sys/module/lowmemorykiller/parameters/minfree
chmod 0664 /sys/module/lowmemorykiller/parameters/minfree

# Tweak background writeout
write /proc/sys/vm/dirty_expire_centisecs 200
write /proc/sys/vm/dirty_background_ratio 5

# Permissions for System Server and daemons.
chown radio system /sys/android_power/state
chown radio system /sys/android_power/request_state
chown radio system /sys/android_power/acquire_full_wake_lock
chown radio system /sys/android_power/acquire_partial_wake_lock
chown radio system /sys/android_power/release_wake_lock
chown system system /sys/power/autosleep
chown system system /sys/power/state
chown system system /sys/power/wakeup_count
chown radio system /sys/power/wake_lock
chown radio system /sys/power/wake_unlock
chmod 0660 /sys/power/state
chmod 0660 /sys/power/wake_lock
chmod 0660 /sys/power/wake_unlock

# Set this property so surfaceflinger is not started by system_init
setprop system_init.startsurfaceflinger 0

class_start core
class_start main

```

和前面对 service 的分析类似, case 中进入 K_on 选项执行函数 parse_action。函数 parse_action 的具体实现代码如下所示:

```

static void *parse_action(struct parse_state *state, int nargs, char **args)
{
    struct action *act;
    if (nargs < 2) {
        parse_error(state, "actions must have a trigger\n");
        return 0;
    }
    if (nargs > 2) {
        parse_error(state, "actions may not have extra parameters\n");
        return 0;
    }
    act = calloc(1, sizeof(*act));
    act->name = args[1];
    list_init(&act->commands);
    list_add_tail(&action_list, &act->alist);
    /* XXX add to hash */
    return act;
}

```

8.5.2 init 组织 on

在 init 进程中可以看到一个名为 action 结构体类似于 service, 这个 action 的名字为 boot, 最

后会将这个 action 加入到全局链表 action_list 中。结构体 action 的具体实现代码如下所示：

```
struct action {
    /* node in list of all actions */
    struct listnode alist;
    /* node in the queue of pending actions */
    struct listnode qlist;
    /* node in list of actions for a trigger */
    struct listnode tlist;

    unsigned hash;
    const char *name;

    struct listnode commands;
    struct command *current;
};
```

8.5.3 解析 on 用到的函数

在解析 on 时会用到函数 parse_line_action，功能是对 on 字段所在的 option 进行解析。函数 parse_line_action 的具体实现代码如下所示：

```
static void parse_line_action(struct parse_state* state, int nargs, char **args)
//action所在的行
{
    struct command *cmd;
    struct action *act = state->context;//on boot 启动
    int (*func)(int nargs, char **args);
    int kw, n;

    if (nargs == 0) {
        return;
    }

    kw = lookup_keyword(args[0]); //命令的参数个数
    if (!kw_is(kw, COMMAND)) {
        parse_error(state, "invalid command '%s'\n", args[0]);
        return;
    }

    n = kw_nargs(kw);
    if (nargs < n) {
        parse_error(state, "%s requires %d %s\n", args[0], n - 1,
            n > 2 ? "arguments" : "argument");
        return;
    }
    cmd = malloc(sizeof(*cmd) + sizeof(char*) * nargs);
    cmd->func = kw_func(kw);
    cmd->nargs = nargs;
    memcpy(cmd->args, args, sizeof(char*) * nargs);
    list_add_tail(&act->commands, &cmd->clist); //
```

到此为止，on 和 service 两个 section 的分析工作全部完成。

8.6 在 init 控制 service

在 Android 系统中，当进行 init 进程初始化的时候，除了对系统做一些必要的初始化外操作，还需要启动 Service。而 Service 是在 init 脚本中定义的，所以很有必要了解一下在 init 中对 service 进行控制的知识，这部分知识将在本节的内容中呈现出来。

8.6.1 启动 Zygote

Android 系统是基于 Linux 内核的，而在 Linux 系统中的所有进程都是 init 进程的子进程或孙

进程。也就是说，所有的进程都是直接或者间接地由 init 进程 fork 出来的。Zygote 进程也不例外，它是在系统启动的过程，由 init 进程创建的。在系统启动脚本文件“system/core/rootdir/init.rc”中，可以看到如下启动 Zygoe 进程的脚本命令：

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
socket zygote stream 666
onrestart write /sys/android_power/request_state wake
onrestart write /sys/power/state on
onrestart restart media
onrestart restart netd
```

在上述代码中，各个关键字的具体说明如下所示。

□ **service**：用于通知 init 进程创建一个名为“Zygote”的进程，这个 Zygoe 进程要执行的程序是“/system/bin/app_process”，后面是要传给 app_process 的参数。

□ **Socket**：表示这个 Zygoe 进程需要一个名称为“Zygote”的 Socket 资源，这样启动系统后，就可以在“/dev/socket”目录下看到有一个名为 Zygoe 的文件。这里定义的 Socket 的类型为 Unix Domain Socket，它是用来作本地进程间通信用的。

8.6.2 启动 service

首先看函数 do_class_start，此函数的功能是启动 service，此函数在如下文件中定义：

```
\system\core\init\builtins.c
```

函数 do_class_start 的具体实现代码如下所示：

```
int do_class_start(int nargs, char **args)
{
    /* Starting a class does not start services
     * which are explicitly disabled. They must
     * be started individually.
     */
    service_for_each_class(args[1], service_start_if_not_disabled);
    return 0;
}
```

在上述代码中，调用了函数 service_start_if_not_disabled 实现启动功能，此函数也在文件 builtins.c 中实现，具体实现代码如下所示：

```
static void service_start_if_not_disabled(struct service *svc)
{
    if (!(svc->flags & SVC_DISABLED)) {
        service_start(svc, NULL);
    }
}
```

在上述代码中，调用了函数 service_start 实现启动功能。函数 service_start 是整个启动功能的核心，在文件 init.c 中定义，具体实现代码如下所示：

```
void service_start(struct service *svc, const char *dynamic_args)
{
    struct stat s;
    pid_t pid;
    int needs_console;
    int n;
#ifndef HAVE_SELINUX
    char *scon = NULL;
    int rc;
#endif
    /* starting a service removes it from the disabled or reset
     * state and immediately takes it out of the restarting
     * state if it was in there
     */
```

```

svc->flags &= (~(SVC_DISABLED|SVC_RESTARTING|SVC_RESET));
svc->time_started = 0;

/* running processes require no additional work -- if
 * they're in the process of exiting, we've ensured
 * that they will immediately restart on exit, unless
 * they are ONESHOT
 */
if (svc->flags & SVC_RUNNING) {
    return;
}

needs_console = (svc->flags & SVC_CONSOLE) ? 1 : 0;
if (needs_console && (!have_console)) {
    ERROR("service '%s' requires console\n", svc->name);
    svc->flags |= SVC_DISABLED;
    return;
}

if (stat(svc->args[0], &s) != 0) {
    ERROR("cannot find '%s', disabling '%s'\n", svc->args[0], svc->name);
    svc->flags |= SVC_DISABLED;
    return;
}

if (((! (svc->flags & SVC_ONESHOT)) && dynamic_args) {
    ERROR("service '%s' must be one-shot to use dynamic args, disabling\n",
          svc->args[0]);
    svc->flags |= SVC_DISABLED;
    return;
}

#endif HAVE_SELINUX
if (is_selinux_enabled() > 0) {
    char *mycon = NULL, *fcon = NULL;

    INFO("computing context for service '%s'\n", svc->args[0]);
    rc = getcon(&mycon);
    if (rc < 0) {
        ERROR("could not get context while starting '%s'\n", svc->name);
        return;
    }

    rc = getfilecon(svc->args[0], &fcon);
    if (rc < 0) {
        ERROR("could not get context while starting '%s'\n", svc->name);
        freecon(mycon);
        return;
    }

    rc = security_compute_create(mycon, fcon, string_to_security_class("process"), &scon);
    freecon(mycon);
    freecon(fcon);
    if (rc < 0) {
        ERROR("could not get context while starting '%s'\n", svc->name);
        return;
    }
}
#endif
NOTICE("starting '%s'\n", svc->name);

pid = fork();

if (pid == 0) {
    struct socketinfo *si;
    struct svcenvinfo *ei;
    char tmp[32];
    int fd, sz;
}

```

```

        umask(077);
#endif __arm__
/*
 * b/7188322 - Temporarily revert to the compat memory layout
 * to avoid breaking third party apps.
 *
 * THIS WILL GO AWAY IN A FUTURE ANDROID RELEASE.
 *
 */
http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=7dbaa466
 * changes the kernel mapping from bottom up to top-down.
 * This breaks some programs which improperly embed
 * an out of date copy of Android's linker.
 */
int current = personality(0xffffffff);
personality(current | ADDR_COMPAT_LAYOUT);
#endif
if (properties_init() ) {
    get_property_workspace(&fd, &sz);
    sprintf(tmp, "%d,%d", dup(fd), sz);
    add_environment("ANDROID_PROPERTY_WORKSPACE", tmp);
}

for (ei = svc->envvars; ei; ei = ei->next)
    add_environment(ei->name, ei->value);

#ifndef HAVE_SELINUX
    setsockcreatecon(scon);
#endif

for (si = svc->sockets; si; si = si->next) {
    int socket_type =
        !strcmp(si->type, "stream") ? SOCK_STREAM :
        (!strcmp(si->type, "dgram") ? SOCK_DGRAM : SOCK_SEQPACKET));
    int s = create_socket(si->name, socket_type,
                          si->perm, si->uid, si->gid);
    if (s >= 0) {
        publish_socket(si->name, s);
    }
}

#ifndef HAVE_SELINUX
    freecon(scon);
    scon = NULL;
    setsockcreatecon(NULL);
#endif

if (svc->ioprio_class != IoSchedClass_NONE) {
    if (android_set_ioprio(getpid(), svc->ioprio_class, svc->ioprio_pri)) {
        ERROR("Failed to set pid %d ioprio = %d,%d: %s\n",
              getpid(), svc->ioprio_class, svc->ioprio_pri, strerror(errno));
    }
}

if (needs_console) {
    setsid();
    open_console();
} else {
    zap_stdio();
}

#if 0
for (n = 0; svc->args[n]; n++) {
    INFO("args[%d] = '%s'\n", n, svc->args[n]);
}
for (n = 0; ENV[n]; n++) {
    INFO("env[%d] = '%s'\n", n, ENV[n]);
}

```

```

#endif

        setpgid(0, getpid());

/* as requested, set our gid, supplemental gids, and uid */
if (svc->gid) {
    if (setgid(svc->gid) != 0) {
        ERROR("setgid failed: %s\n", strerror(errno));
        _exit(127);
    }
}
if (svc->nr_supp_gids) {
    if (setgroups(svc->nr_supp_gids, svc->supp_gids) != 0) {
        ERROR("setgroups failed: %s\n", strerror(errno));
        _exit(127);
    }
}
if (svc->uid) {
    if (setuid(svc->uid) != 0) {
        ERROR("setuid failed: %s\n", strerror(errno));
        _exit(127);
    }
}

#ifndef HAVE_SELINUX
if (svc->seclabel) {
    if (is_selinux_enabled() > 0 && setexeccon(svc->seclabel) < 0) {
        ERROR("cannot setexeccon('%s'): %s\n", svc->seclabel, strerror(errno));
        _exit(127);
    }
}
#endif

if (!dynamic_args) {
    if (execve(svc->args[0], (char**) svc->args, (char**) ENV) < 0) {
        ERROR("cannot execve('%s'): %s\n", svc->args[0], strerror(errno));
    }
} else {
    char *arg_ptrs[INIT_PARSER_MAXARGS+1];
    int arg_idx = svc->nargs;
    char *tmp = strdup(dynamic_args);
    char *next = tmp;
    char *bword;

    /* Copy the static arguments */
    memcpy(arg_ptrs, svc->args, (svc->nargs * sizeof(char *)));

    while((bword = strsep(&next, " "))) {
        arg_ptrs[arg_idx++] = bword;
        if (arg_idx == INIT_PARSER_MAXARGS)
            break;
    }
    arg_ptrs[arg_idx] = '\0';
    execve(svc->args[0], (char**) arg_ptrs, (char**) ENV);
}
_exit(127);
}

#ifndef HAVE_SELINUX
freecon(scon);
#endif

if (pid < 0) {
    ERROR("failed to start '%s'\n", svc->name);
    svc->pid = 0;
    return;
}

svc->time_started = gettimeofday();

```

```

    svc->pid = pid;
    svc->flags |= SVC_RUNNING;

    if (properties_initiated())
        notify_service_state(svc->name, "running");
}

```

在函数 `service_start` 中，参数 `dynamic_args` 只有当 `service` 的 option 中有 `oneshot` 时才会用到，此时会通过替换掉启动服务的命令参数启动服务。因为 `service` 的 option 会记录在 `struct service` 中，所以在启动 `service` 时只需考虑到这些选项即可。同时，会记录下 `service` 的 `pid` 和状态等信息。

8.6.3 4 种启动 service 的方式

其实在 `init` 进程中，可以使用如下所示的方式启动 `service`。

(1) 在 `action` 下面添加和启动服务相关的 `command`，在 `action` 中和操作服务相关的命令如下。

- `class_start <serviceclass> #:` 启动所有指定 `class` 的服务。
- `class_stop <serviceclass> #:` 停止所有指定 `class` 的服务，后续没法通过 `class_start` 启动。
- `class_reset <serviceclass> #:` 停止服务，后续可以通过 `class_start` 启动。
- `restart <servicename> #:` 重启指定名称的服务，先 `stop`，再 `start`。
- `start <servicename> #:` 启动指定名称的服务。
- `stop <servicename> #:` 停止指定名称的服务。

在启动 `command` 时，在文件 `init.c` 的主函数 `main` 中，通过使用 “`for(;;)`” 循环执行函数 `execute_one_command` 的方式实现，此函数的具体实现代码如下所示：

```

void execute_one_command(void)
{
    int ret;

    if (!cur_action || !cur_command || is_last_command(cur_action, cur_command)) {
        cur_action = action_remove_queue_head();
        cur_command = NULL;
        if (!cur_action)
            return;
        INFO("processing action %p (%s)\n", cur_action, cur_action->name);
        cur_command = get_first_command(cur_action);
    } else {
        cur_command = get_next_command(cur_action, cur_command);
    }

    if (!cur_command)
        return;

    ret = cur_command->func(cur_command->nargs, cur_command->args); // 执行 class_start 等
    INFO("command '%s' r=%d\n", cur_command->args[0], ret);
}

```

(2) 使用函数 `restart_processes` 和 `restart_service_if_needed` 重启 Service，该函数位于 `init` 的主线程循环中，功能是查看是否有需要重新启动的 Service。在文件 `init.c` 中，函数 `restart_processes` 和 `restart_service_if_needed` 的具体实现代码如下所示：

```

static void restart_service_if_needed(struct service *svc)
{
    time_t next_start_time = svc->time_started + 5;

    if (next_start_time <= gettimeofday()) {
        svc->flags &= (~SVC_RESTARTING);
        service_start(svc, NULL);
        return;
    }
}

```

```

    if ((next_start_time < process_needs_restart) ||
        (process_needs_restart == 0)) {
        process_needs_restart = next_start_time;
    }
}

static void restart_processes()
{
    process_needs_restart = 0;
    service_for_each_flags(SVC_RESTARTING,
                           restart_service_if_needed);
}

```

在重启过程中，会重启 flag 为 SVC_RESTARTING 的服务。这部分进程的重启其实在 init 由 handle_signal 来管理，一旦出现 Service 崩溃，函数 poll 会接收到相关文件变化的信息，并执行 handle_signal 中的函数 wait_for_one_process。函数 wait_for_one_process 的具体实现代码如下所示：

```

static int wait_for_one_process(int block)
{
    pid_t pid;
    int status;
    struct service *svc;
    struct socketinfo *si;
    time_t now;
    struct listnode *node;
    struct command *cmd;

    while ( (pid = waitpid(-1, &status, block ? 0 : WNOHANG)) == -1 && errno == EINTR );
    if (pid <= 0) return -1;
    INFO("waitpid returned pid %d, status = %08x\n", pid, status);

    svc = service_find_by_pid(pid);
    if (!svc) {
        ERROR("untracked pid %d exited\n", pid);
        return 0;
    }

    ...
    svc->flags |= SVC_RESTARTING;

    /* Execute all onrestart commands for this service. */
    list_for_each(node, &svc->onrestart.commands) {
        cmd = node_to_item(node, struct command, clist);
        cmd->func(cmd->nargs, cmd->args);
    }
    notify_service_state(svc->name, "restarting");
    return 0;
}

```

在上述代码中，使用 waitpid 找到子进程退出的进程号 pid，然后查找到该 Service，对 Service 中的 onrestart 这个 commands 进行操作。同时将 Service 的 flag 设置为 SVC_RESTARTING，这样就结合前面讲到的 restart_processes 便重新启动了该服务进程。

(3) 在文件 “\system\core\init\property_service.c” 中，使用函数 handle_property_set_f 向 socket 中名称为 property_service 的属性服务发送控制的消息，这样便可以进入到该函数中。函数 handle_property_set_f 的具体实现代码如下所示：

```

void handle_property_set_fd()
{
    prop_msg msg;
    int s;
    int r;
    int res;
    struct ucred cr;
    struct sockaddr_un addr;
    socklen_t addr_size = sizeof(addr);
    socklen_t cr_size = sizeof(cr);

```

```

char * source_ctx = NULL;

if ((s = accept(property_set_fd, (struct sockaddr *) &addr, &addr_size)) < 0) {
    return;
}

/* Check socket options here */
if (getsockopt(s, SOL_SOCKET, SO_PEERCRED, &cr, &cr_size) < 0) {
    close(s);
    ERROR("Unable to receive socket options\n");
    return;
}

r = TEMP_FAILURE_RETRY(recv(s, &msg, sizeof(msg), 0));
if(r != sizeof(prop_msg)) {
    ERROR("sys_prop: mis-match msg size received: %d expected: %d errno: %d\n",
          r, sizeof(prop_msg), errno);
    close(s);
    return;
}

switch(msg.cmd) {
case PROP_MSG_SETPROP:
    msg.name[PROP_NAME_MAX-1] = 0;
    msg.value[PROP_VALUE_MAX-1] = 0;

#ifdef HAVE_SELINUX
    getpeercon(s, &source_ctx);
#endif

    if(memcmp(msg.name,"ctl.",4) == 0) {
        // Keep the old close-socket-early behavior when handling
        // ctl.* properties.
        close(s);
        if (check_control_perms(msg.value, cr.uid, cr.gid, source_ctx)) {
            handle_control_message((char*) msg.name + 4, (char*) msg.value);
        } else {
            ERROR("sys_prop: Unable to %s service ctl [%s] uid:%d gid:%d pid:%d\n",
                  msg.name + 4, msg.value, cr.uid, cr.gid, cr.pid);
        }
    } else {
        if (check_perms(msg.name, cr.uid, cr.gid, source_ctx)) {
            property_set((char*) msg.name, (char*) msg.value);
        } else {
            ERROR("sys_prop: permission denied uid:%d name:%s\n",
                  cr.uid, msg.name);
        }
    }

    // Note: bionic's property client code assumes that the
    // property server will not close the socket until *AFTER*
    // the property is written to memory.
    close(s);
}
#endif HAVE_SELINUX
freecon(source_ctx);
#endif

break;

default:
    close(s);
    break;
}
}

```

(4) 使用函数 `handle_keychord` 启动，该函数和 `chorded keyboard` 有关，功能是处理注册在 Service structure 上的 keychord，通常是启动 Service。函数 `handle_keychord` 在文件“`system/core/init/keychords.c`”中定义，具体实现代码如下所示：

```

void handle_keychord()
{
    struct service *svc;
    const char* debuggable;
    const char* adb_enabled;
    int ret;
    __u16 id;

    // only handle keychords if ro.debuggable is set or adb is enabled.
    // the logic here is that bugreports should be enabled in userdebug or eng builds
    // and on user builds for users that are developers.
    debuggable = property_get("ro.debuggable");
    adb_enabled = property_get("init.svc.adbd");
    ret = read(keychord_fd, &id, sizeof(id));
    if (ret != sizeof(id)) {
        ERROR("could not read keychord id\n");
        return;
    }

    if ((debuggable && !strcmp(debuggable, "1")) ||
        (adb_enabled && !strcmp(adb_enabled, "running")))
    {
        svc = service_find_by_keychord(id);
        if (svc) {
            INFO("starting service %s from keychord\n", svc->name);
            service_start(svc, NULL);
        } else {
            ERROR("service for keychord %d not found\n", id);
        }
    }
}

```

8.7 控制属性服务

编写过 Windows 本地应用的读者应该知道，在 Windows 中的注册表机制中提供了大量的属性，其实在 Linux 中也有类似的机制：属性服务。init 在启动的过程中会启动属性服务（Socket 服务），并且在内存中建立一块存储区域，用来存储这些属性。当读取这些属性时，直接从这一内存区域读取，如果修改属性值，需要通过 Socket 连接属性服务完成。在文件 init.c 中，函数 action 调用了函数 start_property_service 来启动属性服务，action 是 init.rc 及其类似文件中的一种执行机制。

在文件 init.c 中，可以看到和属性操作相关的代码情景，例如：

```

property_init()
property_set_fd

```

在本节的内容中，将详细讲解在 Android 4.3 系统中 init 控制属性服务的基本知识。

8.7.1 引入属性

在文件 init.c 的主函数 main 中，调用函数 property_init 为属性分配了一些存储空间。如果查看文件 init.rc，会发现该文件开始部分用一些 import 语句导入了其他的配置文件，例如，/init.usb.rc。大多数配置文件都直接使用了确定的文件名，只有如下的代码使用了一个变量（\${ro.hardware}）执行了配置文件名的一部分。

```

import /init.${ro.hardware}.rc

```

要想了解上述变量的获得过程，首先需要了解配置文件 init.\${ro.hardware}.rc 的内容，这些通常与当前的硬件有关，其中函数 get_hardware_name 用于获取硬件的名称信息，具体代码如下所示：

```

void get_hardware_name(char *hardware, unsigned int *revision)
{
    char data[1024];

```

```

int fd, n;
char *x, *hw, *rev;

/* 如果 hardware 已经有值了，说明 hardware 通过内核命令行提供，直接返回 */
if (hardware[0])
    return;
// 打开/proc/cpuinfo 文件
fd = open("/proc/cpuinfo", O_RDONLY);
if (fd < 0) return;
// 读取/proc/cpuinfo 文件的内容
n = read(fd, data, 1023);
close(fd);
if (n < 0) return;

data[n] = 0;
// 从/proc/cpuinfo 文件中获取 Hardware 字段的值
hw = strstr(data, "\nHardware");
rev = strstr(data, "\nRevision");
// 成功获取 Hardware 字段的值
if (hw) {
    x = strstr(hw, ": ");
    if (x) {
        x += 2;
        n = 0;
        while (*x && *x != '\n') {
            if (!isalpha(*x))
                // 将 Hardware 字段的值都转换为小写，并更新 hardware 参数的值
                // hardware 也就是在 init.c 文件中定义的 hardware 数组
                hardware[n++] = tolower(*x);
            x++;
            if (n == 31) break;
        }
        hardware[n] = 0;
    }
}
if (rev) {
    x = strstr(rev, ": ");
    if (x) {
        *revision = strtoul(x + 2, 0, 16);
    }
}
}

```

从上述代码可以得知，该函数主要用于确定 hardware 和 revision 的变量的值。获取 hardware 的来源是从 Linux 内核命令行或文件 “/proc/cpuinfo” 中的内容，文件 “/proc/cpuinfo” 是虚拟文件（内存文件），执行 cat /proc/cpuinfo 命令会看到该文件中的内容，如图 8-1 所示。

```

Processor      : ARMv7 Processor rev 9 (v7l)
processor     : 0
BogoMIPS      : 1993.93

processor     : 1
BogoMIPS      : 1993.93

processor     : 2
BogoMIPS      : 1993.93

processor     : 3
BogoMIPS      : 1993.93

Features      : swp half thumb fastmult vfp edsp neon vfpv3 tls
CPU implementer: 0x41
CPU architecture: 7
CPU variant   : 0x2
CPU part      : 0xc09
CPU revision  : 9

Hardware      : grouper
Revision      : 0000
Serial        : 0f410a0001440200
root@android:/ #

```

图 8-1 显示文件内容

在图 8-1 中, 白框中的内容就是 Hardware 字段的值。由于该设备是 Nexus 7, 所以值为 grouper。如果程序就到此位置, 那么与硬件有关的配置文件名是 init.grouper.rc。有 Nexus 7 的读者会看到在根目录下确实有一个 init.grouper.rc 文件。说明 Nexus 7 的原生 ROM 并没有在其他的地方设置配置文件名, 所以配置文件名就是从 “/proc/cpuinfo” 文件的 Hardware 字段中取的值。

接下来看在函数 `get_hardware_name` 后面调用的函数 `process_kernel_cmdline`, 具体实现代码如下所示:

```
static void process_kernel_cmdline(void)
{
    /* don't expose the raw commandline to nonpriv processes */
    chmod("/proc/cmdline", 0440);

    // 导入内核命令行参数
    import_kernel_cmdline(0, import_kernel_nv);
    if (qemu[0])
        import_kernel_cmdline(1, import_kernel_nv);

    // 用属性值设置内核变量
    export_kernel_boot_props();
}
```

在上述代码中, 除了使用函数 `import_kernel_cmdline` 导入内核变量外, 其主要功能是调用函数 `export_kernel_boot_props` 通过属性设置内核变量。例如, 通过属性 `ro.boot.hardware` 设置 `hardware` 变量。也就是说, 可以通过属性值 `ro.boot.hardware` 修改函数 `get_hardware_name` 中从文件 “/proc/cpuinfo” 得到的 `hardware` 字段值。函数 `export_kernel_boot_props` 的具体实现代码如下所示:

```
static void export_kernel_boot_props(void)
{
    char tmp[PROP_VALUE_MAX];
    const char *pval;
    unsigned i;
    struct {
        const char *src_prop;
        const char *dest_prop;
        const char *def_val;
    } prop_map[] = {
        { "ro.boot.serialno", "ro.serialno", "" },
        { "ro.boot.mode", "ro.bootmode", "unknown" },
        { "ro.boot.baseband", "ro.baseband", "unknown" },
        { "ro.boot.bootloader", "ro.bootloader", "unknown" },
    };
    // 通过内核的属性设置应用层配置文件的属性
    for (i = 0; i < ARRAY_SIZE(prop_map); i++) {
        pval = property_get(prop_map[i].src_prop);
        property_set(prop_map[i].dest_prop, pval ?: prop_map[i].def_val);
    }
    // 根据 ro.boot.console 属性的值设置 console 变量
    pval = property_get("ro.boot.console");
    if (pval)
        strlcpy(console, pval, sizeof(console));

    /* save a copy for init's usage during boot */
    strlcpy(bootmode, property_get("ro.bootmode"), sizeof(bootmode));

    /* if this was given on kernel command line, override what we read
     * before (e.g. from /proc/cpuinfo), if anything */
    // 获取 ro.boot.hardware 属性的值
    pval = property_get("ro.boot.hardware");
    if (pval)
        // 这里通过 ro.boot.hardware 属性再次改变 hardware 变量的值
        strlcpy(hardware, pval, sizeof(hardware));
    // 利用 hardware 变量的值设置设置 ro.hardware 属性
    // 这个属性就是前面提到的设置初始化文件名的属性, 实际上是通过 hardware 变量设置的
    property_set("ro.hardware", hardware);
}
```

```

snprintf(tmp, PROP_VALUE_MAX, "%d", revision);
property_set("ro.revision", tmp);

/* TODO: these are obsolete. We should delete them */
if (!strcmp(bootmode,"factory"))
    property_set("ro.factorytest", "1");
else if (!strcmp(bootmode,"factory2"))
    property_set("ro.factorytest", "2");
else
    property_set("ro.factorytest", "0");
}

```

由上述代码可以看出，该函数实际上就是来回设置一些属性值，并且利用某些属性值修改 console、hardware 等变量。其中 hardware 变量（就是一个长度为 32 的字符数组）在函数 get_ hardware_name 中已经从文件 “/proc/cpuinfo” 中获得过一次值了，在函数 export_kernel_ boot_props 中又通过属性 ro.boot.hardware 设置了一次值，不过在 Nexus 7 中并没有设置该属性，所以 hardware 的值仍为 grouper。最后用变量 hardware 设置属性 ro.hardware，所以最后的初始化文件名为 init.grouper.rc。

8.7.2 初始化属性服务

在文件 “\system\core\init\property_service.c” 中，使用函数 property_init 初始化属性存储区域，具体实现代码如下所示：

```

void property_init(void)
{
    init_property_area();
}

```

在上述代码中，函数 init_property_area 也是在文件 property_service.c 中实现的，该函数用于初始化属性内存区域，也就是 __system_property_area__ 变量。函数 init_property_area 的具体实现代码如下所示：

```

static int init_property_area(void)
{
    prop_area *pa;

    if(pa_info_array)
        return -1;

    if(init_workspace(&pa_workspace, PA_SIZE))
        return -1;

    fcntl(pa_workspace.fd, F_SETFD, FD_CLOEXEC);

    pa_info_array = (void*) (((char*) pa_workspace.data) + PA_INFO_START);

    pa = pa_workspace.data;
    memset(pa, 0, PA_SIZE);
    pa->magic = PROP_AREA_MAGIC;
    pa->version = PROP_AREA_VERSION;

    /* plug into the lib property services */
    __system_property_area__ = pa;
    property_area_initied = 1;
    return 0;
}

```

8.7.3 启动属性服务

在文件 “\system\core\init\property_service.c” 中，使用函数 start_property_service 启动一个属性服务器，具体实现代码如下所示：

```

void start_property_service(void)
{
    int fd;
    // 装载不同的属性文件
    load_properties_from_file(PROP_PATH_SYSTEM_BUILD);
    load_properties_from_file(PROP_PATH_SYSTEM_DEFAULT);
    load_override_properties();
    /* Read persistent properties after all default values have been loaded. */
    load_persistent_properties();
    // 创建 socket 服务 (属性服务)
    fd = create_socket(PROP_SERVICE_NAME, SOCK_STREAM, 0666, 0, 0);
    if(fd < 0) return;
    fcntl(fd, F_SETFD, FD_CLOEXEC);
    fcntl(fd, F_SETFL, O_NONBLOCK);
    // 开始服务监听
    listen(fd, 8);
    property_set_fd = fd;
}

```

通过上述代码可以知道属性服务的启动方式，另外在函数 `start_property_service` 中还涉及如下所示的两个宏。

- `PROP_PATH_SYSTEM_BUILD`。
- `PROP_PATH_SYSTEM_DEFAULT`。

上述两个宏都是系统预定义的属性文件名的路径，为了获取这些宏的定义，需要先分析函数 `property_get`，该函数也是在 `Property_service.c` 中实现，具体实现代码如下所示：

```

const char* property_get(const char *name)
{
    prop_info *pi;
    if(strlen(name) >= PROP_NAME_MAX) return 0;
    pi = (prop_info*) __system_property_find(name);
    if(pi != 0) {
        return pi->value;
    } else {
        return 0;
    }
}

```

通过上述代码可以看到，在函数 `property_get` 中调用了核心函数 `__system_property_find`，该核心函数真正实现了获取属性值的功能。函数 `__system_property_find` 属于 bionic 的一个 library，在文件 `system_properties.c` 中实现，可以在如下的目录找到该文件。

/bionic/libc/bionic

函数 `__system_property_find` 的具体实现代码如下所示：

```

const prop_info * __system_property_find(const char *name)
{
    // 获取属性存储内存区域的首地址
    prop_area *pa = __system_property_area__;
    unsigned count = pa->count;
    unsigned *toc = pa->toc;
    unsigned len = strlen(name);
    prop_info *pi;

    while(count--) {
        unsigned entry = *toc++;
        if(TOC_NAME_LEN(entry) != len) continue;

        pi = TOC_TO_INFO(pa, entry);
        if(memcmp(name, pi->name, len)) continue;
        return pi;
    }
    return 0;
}

```

从上述函数 `_system_property_find` 的实现代码可以看出，第一行使用了一个 `_system_property_area` 变量，该变量是全局的。

在文件 `system_properties.c` 对应的头文件 `system_properties.h` 中，定义了前面提到的两个表示属性文件路径的宏，其实还有另外两个表示路径的宏，共 4 个属性文件。文件 `system_properties.h` 可以在如下所示的目录中找到

```
/bionic/libc/include/sys
```

这 4 个宏的具体定义如下所示：

```
#define PROP_PATH_RAMDISK_DEFAULT "/default.prop"
#define PROP_PATH_SYSTEM_BUILD "/system/build.prop"
#define PROP_PATH_SYSTEM_DEFAULT "/system/default.prop"
#define PROP_PATH_LOCAL_OVERRIDE "/data/local.prop"
```

此时可以进入 Android 设备的相应目录找到上述 4 个文件，一般会被保存在根目录中，通常在文件 `default.prop` 和 `catdefault.prop` 中会看到该文件的内容。而属性服务就是装载所有这 4 个属性文件中的所有属性以及使用 `property_set` 设置的属性。在 Android 设备的终端可以直接使用 `getprop` 命令从属性服务获取所有的属性值，如图 8-2 所示。另外，`getprop` 命令还可以直接根据属性名还获取具体的属性值，例如：

```
getprop ro.build.product
```

```
root@android:/dev/socket # getprop
[camera.flash.off]: [0]
[dalvik.vm.dexopt.flags]: [xyz]
[dalvik.vm.heapgrowthlimit]: [8m]
[dalvik.vm.heapmaxfree]: [8m]
[dalvik.vm.heapminfree]: [512k]
[dalvik.vm.heapsize]: [384m]
[dalvik.vm.heaptargetutilization]: [0.75]
[dalvik.vm.stack-trace-file]: [/data/anr/traces.txt]
[debug.nfc.fw.download]: [false]
[debug.nfc.se]: [false]
[dev.bootcomplete]: [1]
[dhcp.wlan0.dns1]: [192.168.17.254]
[dhcp.wlan0.dns2]: []
[dhcp.wlan0.dns3]: []
[dhcp.wlan0.dns4]: []
[dhcp.wlan0.gateway]: [192.168.17.254]
[dhcp.wlan0.ipaddress]: [192.168.17.104]
[dhcp.wlan0.leasetime]: [7200]
[dhcp.wlan0.mask]: [255.255.255.0]
[dhcp.wlan0.ptid]: [10749]
[dhcp.wlan0.reason]: [IRENEW]
[dhcp.wlan0.result]: [ok]
[dhcp.wlan0.server]: [192.168.17.254]
```

图 8-2 从属性服务获取所有的属性值

在 Android 4.3 源码中，`getprop` 命令的源代码文件是 `getprop.c`。读者可以在如下所示的目录中找到该文件：

```
/system/core/toolbox/
```

其实 `getprop` 获取属性值也是通过函数 `property_get` 完成的，此函数实际上调用了函数 `_system_property_find`，从 `_system_property_area` 变量指定的内存区域获取相应的属性值。另外，在文件 `system_properties.c` 中还有如下两个函数用于通过属性服务修改或添加某个属性的值。

```
static int send_prop_msg(prop_msg *msg)
{
    struct pollfd pollfds[1];
    struct sockaddr_un addr;
    socklen_t alen;
    size_t namelen;
    int s;
    int r;
    int result = -1;
    // 创建用于连接属性服务的 socket
    s = socket(AF_LOCAL, SOCK_STREAM, 0);
    if(s < 0) {
        return result;
```

```

}

memset(&addr, 0, sizeof(addr));
// property_service_socket 是 Socket 设备文件名称
namelen = strlen(property_service_socket);
strlcpy(addr.sun_path, property_service_socket, sizeof(addr.sun_path));
addr.sun_family = AF_LOCAL;
alen = namelen + offsetof(struct sockaddr_un, sun_path) + 1;

if(TEMP_FAILURE_RETRY(connect(s, (struct sockaddr *) &addr, alen)) < 0) {
    close(s);
    return result;
}

r = TEMP_FAILURE_RETRY(send(s, msg, sizeof(prop_msg), 0));

if(r == sizeof(prop_msg)) {
    pollfds[0].fd = s;
    pollfds[0].events = 0;
    r = TEMP_FAILURE_RETRY(poll(pollfds, 1, 250 /* ms */));
    if (r == 1 && (pollfds[0].revents & POLLHUP) != 0) {
        result = 0;
    } else {

        result = 0;
    }
}

close(s);
return result;
}
// 用户可以直接调用该函数设置属性值
int __system_property_set(const char *key, const char *value)
{
    int err;
    int tries = 0;
    int update_seen = 0;
    prop_msg msg;

    if(key == 0) return -1;
    if(value == 0) value = "";
    if(strlen(key) >= PROP_NAME_MAX) return -1;
    if(strlen(value) >= PROP_VALUE_MAX) return -1;

    memset(&msg, 0, sizeof msg);
    msg.cmd = PROP_MSG_SETPROP;
    strlcpy(msg.name, key, sizeof msg.name);
    strlcpy(msg.value, value, sizeof msg.value);
    // 设置属性值
    err = send_prop_msg(&msg);
    if(err < 0) {
        return err;
    }
    return 0;
}
}

```

在函数 `send_prop_msg` 中，涉及了重要变量 `property_service_socket`，具体定义如下所示：

```
static const char property_service_socket[] = "/dev/socket/" PROP_SERVICE_NAME;
```

实际上，`send_prop_msg` 通过这个设备文件与属性服务通信。读者可以在 Android 设备的终端进入 “/dev/socket” 目录，通常会看到一个名为 `property_service` 的文件，该文件就是属性服务映射的设备文件。

8.7.4 处理设置属性的请求

当属性服务器收到客户端的请求时，init 会调用函数 `handle_property_set_fd` 进行处理。当客

客户端的权限满足要求时，init 就调用函数 property_set 进行相关的处理。

```

int property_set(const char *name, const char *value)
{
    prop_area *pa;
    prop_info *pi;

    int namelen = strlen(name);
    int valuelen = strlen(value);

    if(namelen >= PROP_NAME_MAX) return -1;
    if(valuelen >= PROP_VALUE_MAX) return -1;
    if(namelen < 1) return -1;

    pi = (prop_info*) __system_property_find(name);

    if(pi != 0) {
        /* ro.* properties may NEVER be modified once set */
        if(!strncmp(name, "ro.", 3)) return -1;

        pa = __system_property_area__;
        update_prop_info(pi, value, valuelen);
        pa->serial++;
        __futex_wake(&pa->serial, INT32_MAX);
    } else {
        pa = __system_property_area__;
        if(pa->count == PA_COUNT_MAX) return -1;

        pi = pa_info_array + pa->count;
        pi->serial = (valuelen << 24);
        memcpy(pi->name, name, namelen + 1);
        memcpy(pi->value, value, valuelen + 1);

        pa->toc[pa->count] =
            (namelen << 24) | (((unsigned) pi) - ((unsigned) pa));

        pa->count++;
        pa->serial++;
        __futex_wake(&pa->serial, INT32_MAX);
    }

    /* If name starts with "net." treat as a DNS property. */
    if (strncmp("net.", name, strlen("net.")) == 0) {
        if (strcmp("net.change", name) == 0) {
            return 0;
        }
        /*
         * The 'net.change' property is a special property used track when any
         * 'net.*' property name is updated. It is _ONLY_ updated here. Its value
         * contains the last updated 'net.*' property.
         */
        property_set("net.change", name);
    } else if (persistent_properties_loaded &&
               strncmp("persist.", name, strlen("persist.")) == 0) {
        /*
         * Don't write properties to disk until after we have read all default properties
         * to prevent them from being overwritten by default values.
         */
        write_persistent_property(name, value);
    #ifdef HAVE_SELINUX
    } else if (strcmp("selinux.reload_policy", name) == 0 &&
               strcmp("1", value) == 0) {
        selinux_reload_policy();
    #endif
    }
    property_changed(name, value);
    return 0;
}

```

到此为止，整个属性服务器的源码知识就介绍完毕了。

第9章 Dalvik VM的进程系统

在Android系统中，存在如下3个十分重要的进程系统。

- Zygote进程：被称为“孵化”进程或“孕育”进程，功能和Linux系统的fork类似，用于“孕育”产生出不同的子进程。
- System进程：是系统进程，是整个Android Framework所在的进程，用于启动Android系统。其核心进程是system_server，它的父进程就是Zygote。
- 应用程序进程：每个Android应用程序运行后都会拥有自己的进程，这和Windows资源管理器中的体现的进程是同一个含义。

在本章的内容中，将详细分析Android L中上述3大进程系统的基本知识，彻底深入了解Android进程系统的方方面面。

9.1 Zygote（孕育）进程详解

在本书前面的内容中，已经了解过Zygote进程的一些知识。在Android系统中，如果查看进程列表，会发现进程Zygote的父进程是init，而且它是所有应用的父进程；还有一个进程是system_server，它的父进程是Zygote。Zygote服务实际上是一种Select服务模型，是为启动Java代码而生，完成了一次androidRuntime的打开和关闭操作。

9.1.1 Zygote基础

Android系统是基于Linux内核的，在Linux系统中的所有进程都是直接或者间接地由init进程fork（孕育）出来的，Zygote进程也不例外。Zygote进程是在系统启动的过程由init进程创建的，是Android系统的核心进程之一，被认为是Android framework大家族的祖先。事实上，Zygote正是平常所说的Java运行环境（JVM）。从总体架构上看，Zygote是一个简单的典型C/S结构。其他进程作为一个客服端向Zygote发出“孕育”请求，当Zygote接收到命令后就“孕育”出一个Activity进程。具体“孕育”过程如图9-1所示。

在Android系统中，Zygote本身是一个应用层的程序，和驱动、内核模块等没有任何关系。Zygote系统源码的组成及其调用结构如下所示。

（1）Zygote.java。

提供访问Dalvik的“zygote”接口，主要是包装Linux系统的Fork（孕育），以建

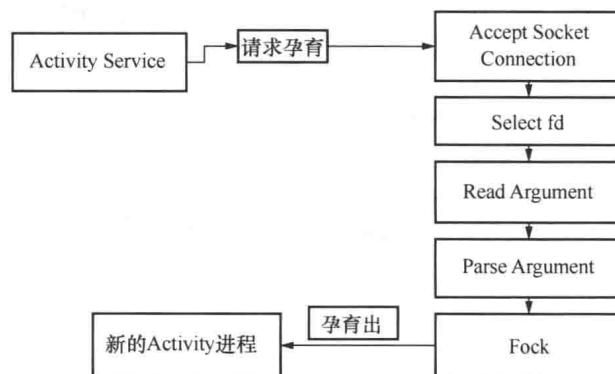


图9-1 Zygote的“孕育”过程